CHAPTER 20

API Controllers

Not all controllers are used to send HTML documents to clients. There are also API controllers, which are used to provide access to an application's data. This is a feature that was previously provided through the separate Web API framework but has now been integrated into ASP.NET Core MVC. In this chapter, I explain the role that API controllers play in a web application, describe the problems they solve, and demonstrate how they are created, tested, and used. Table 20-1 puts API controllers in context.

Table 20-1. *Putting API Controllers in Context*

Question	Answer
What are they?	API controllers are like regular controllers, except that the responses produced by their action methods are data objects that are sent to the client without HTML markup.
Why are they useful?	API controllers allow clients to access the data in an application without also receiving the HTML markup that is required to present that content to the user. Not all clients are browsers, and not all clients present data to a user. An API controller makes an application open for supporting new types of clients or clients developed by a third-party.
How are they used?	API controllers are used like regular HTML controllers.
Are there any pitfalls or limitations?	The most common problems relate to the way that data objects are serialized so they can be sent to the client. See the "Understanding Content Formatting" section for details.
Are there any alternatives?	You don't have to use API controllers in your project, but doing so can increase the value of your platform to your clients.

Table 20-2 summarizes the chapter.

Table 20-2. *Chapter Summary*

Problem	Solution	Listing
Provide access to the data in an application	Create an API controller	10
Request data from an API controller	Use an Ajax query, either directly using the browser's API or through a library like jQuery	11–13
Override the content negotiation process	Use the `Produces` attribute	14-16
Allow clients to override the `Accept` header by specifying the data format in the URL	Add formatter mappings in the `Startup` class, add a segment variable that captures the data format, and, optionally, apply the `FormatFilter` attribute	17–18
Provide full support for the content negotiation process	Enable the `HttpNotAcceptableOutputFormatter` formatter and set the `RespectBrowserAcceptHeader` configuration property	19–20
Receive data in different formats using different action methods	Apply the `Consumes` attribute	21

Preparing the Example Project

For this chapter, I used the ASP.NET Core Web Application (.NET Core) template to create a new Empty project called ApiControllers.

Creating the Model and Repository

I started by creating the `Models` folder, adding a class file called `Reservation.cs`, and using it to define the model class shown in Listing 20-1.

Listing 20-1. The Contents of the Reservation.cs File in the Models Folder

```
namespace ApiControllers.Models {

    public class Reservation {
        public int ReservationId { get; set; }
        public string ClientName { get; set; }
        public string Location { get; set; }
    }
}
```

I also added a file called `IRepository.cs` to the `Models` folder and used it to define the interface for a model repository, as shown in Listing 20-2.

Listing 20-2. The Contents of the IRepository.cs File in the Models Folder

```
using System.Collections.Generic;

namespace ApiControllers.Models {

    public interface IRepository {

        IEnumerable<Reservation> Reservations { get; }
        Reservation this[int id] { get; }

        Reservation AddReservation(Reservation reservation);
        Reservation UpdateReservation(Reservation reservation);
        void DeleteReservation(int id);
    }
}
```

I added a class file called `MemoryRepository.cs` to the `Models` folder and used it to define a nonpersistent implementation of the `IRepository` interface, as shown in Listing 20-3.

Listing 20-3. The Contents of the MemoryRepository.cs File in the Models Folder

```
using System.Collections.Generic;

namespace ApiControllers.Models {

    public class MemoryRepository : IRepository {
        private Dictionary<int, Reservation> items;

        public MemoryRepository() {
            items = new Dictionary<int, Reservation>();
            new List<Reservation> {
                new Reservation { ClientName = "Alice", Location = "Board Room" },
                new Reservation { ClientName = "Bob", Location = "Lecture Hall" },
                new Reservation { ClientName = "Joe", Location = "Meeting Room 1" }
            }.ForEach(r => AddReservation(r));
        }

        public Reservation this[int id] => items.ContainsKey(id) ? items[id] : null;

        public IEnumerable<Reservation> Reservations => items.Values;

        public Reservation AddReservation(Reservation reservation) {
            if (reservation.ReservationId == 0) {
                int key = items.Count;
                while (items.ContainsKey(key)) { key++; };
                reservation.ReservationId = key;
            }
            items[reservation.ReservationId] = reservation;
            return reservation;
        }

        public void DeleteReservation(int id) => items.Remove(id);

        public Reservation UpdateReservation(Reservation reservation)
            => AddReservation(reservation);

    }
}
```

The repository creates a simple set of model objects when it is instantiated, and since there is no persistent storage, any changes will be lost when the application is stopped or restarted. (See Chapter 8 for an example of how to create a persistent repository as part of the SportsStore example application.)

Creating the Controller and Views

Later in the chapter, I will be creating RESTful controllers, but in preparation, I need to create a regular controller to provide a foundation for later examples. I created the `Controllers` folder, added a file called `HomeController.cs`, and used it to define the controller shown in Listing 20-4.

Listing 20-4. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    public class HomeController : Controller {
        private IRepository repository { get; set; }

        public HomeController(IRepository repo) => repository = repo;

        public ViewResult Index() => View(repository.Reservations);

        [HttpPost]
        public IActionResult AddReservation(Reservation reservation) {
            repository.AddReservation(reservation);
            return RedirectToAction("Index");
        }
    }
}
```

This controller defines the `Index` action, which is the default for the application and renders the data model. It also defines an `AddReservation` action, which is accessible only for HTTP `POST` requests and will be used to receive form data from the user. These actions follow the Post/Redirect/Get pattern described in Chapter 17 so that reloading the web page won't create a duplicate form submission.

I created a layout so that I can separate the HTML content from the document header, which will simplify some changes I make later in the chapter. I created the `Views/Shared` folder, added a layout called the `_Layout.cshtml` file, and added the markup shown in Listing 20-5.

Listing 20-5. The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RESTful Controllers</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
</head>
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

Next, I created the `Views/Home` folder, added a view file called `Index.cshtml`, and added the content shown in Listing 20-6.

Listing 20-6. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model IEnumerable<Reservation>
@{  Layout = "_Layout"; }

<form id="addform" asp-action="AddReservation" method="post">
    <div class="form-group">
        <label for="ClientName">Name:</label>
        <input class="form-control" name="ClientName" />
    </div>
    <div class="form-group">
        <label for="Location">Location:</label>
        <input class="form-control" name="Location" />
    </div>
    <div class="text-center panel-body">
        <button type="submit" class="btn btn-sm btn-primary">Add</button>
    </div>
</form>

<table class="table table-sm table-striped table-bordered m-2">
    <thead><tr><th>ID</th><th>Client</th><th>Location</th></tr></thead>
    <tbody>
        @foreach (var r in Model) {
            <tr>
                <td>@r.ReservationId</td>
                <td>@r.ClientName</td>
                <td>@r.Location</td>
            </tr>
        }
    </tbody>
</table>
```

This strongly typed view receives a sequence of `Reservation` objects as its model and uses a Razor `foreach` loop to populate a table with them. There is also a `form` that has been configured to send `POST` requests to the `AddReservation` action.

The examples in this chapter depend on the Bootstrap CSS package. To add Bootstrap to the project, I used the Bower Configuration File item template to create the `bower.json` file in the root of the project and added the package to the `dependencies` section, as shown in Listing 20-7.

Listing 20-7. Adding a Package in the bower.json File

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Next, I created a `_ViewImports.cshtml` file in the `Views` folder and used it to set up the built-in tag helpers for use in Razor views and to import the model namespace, as shown in Listing 20-8.

Listing 20-8. The Contents of the _ViewImports.cshtml File in the Views Folder

```
@using ApiControllers.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

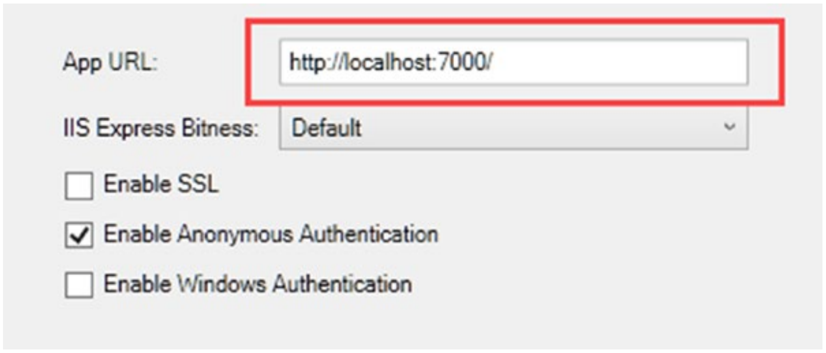
To enable the MVC Framework and the middleware components required for development, I made the changes shown in Listing 20-9 to the `Startup` class. I also used the `AddSingleton` method to set up the service mapping for the model repository.

Listing 20-9. Enabling Middleware in the Startup.cs File in the ApiControllers Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IRepository, MemoryRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

Setting the HTTP Port

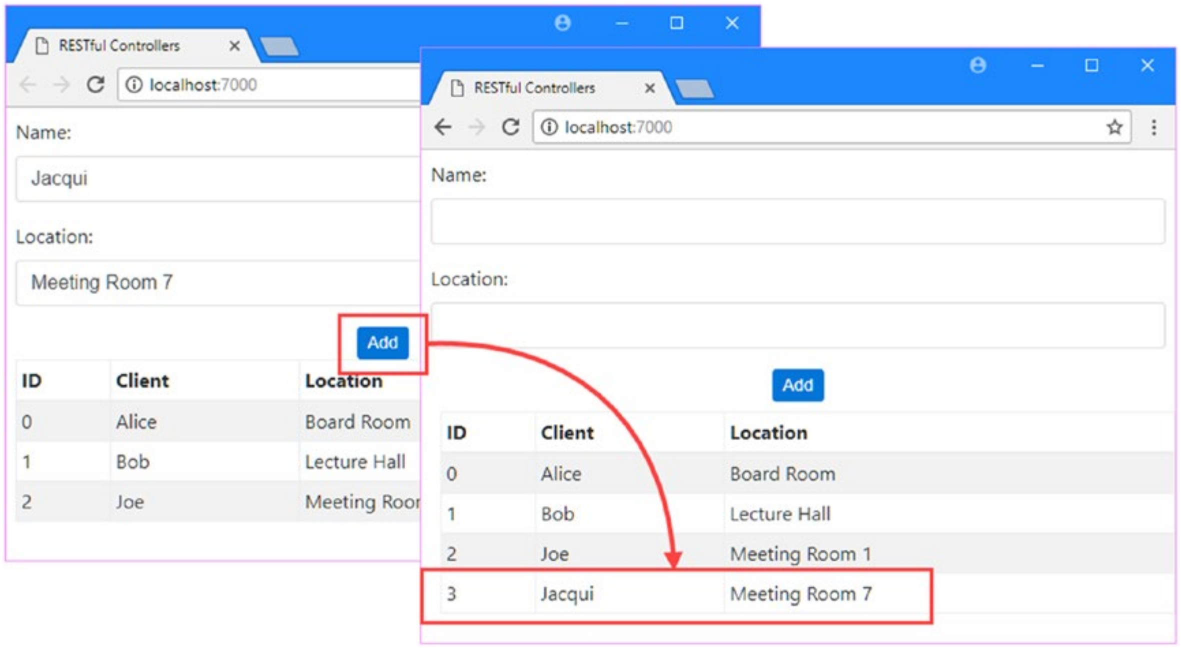
Some of the examples in this chapter are tested by manually typing URLs. To make this easier to describe, I will set the port that will be used to receive HTTP requests. Select ApiControllers Properties from the Visual Studio Project menu, display the Debug tab, and change the value of the App URL field to `http://localhost:7000/`, as shown in Figure 20-1. Make sure you save the changes after you have set the port number.



App URL: http://localhost:7000/
IIS Express Bitness: Default
☐ Enable SSL
☑ Enable Anonymous Authentication
☐ Enable Windows Authentication

Figure 20-1. *Setting the application URL*

Start the application, fill out the form, and click the Add button; the application will add a new `Reservation` to the model, as shown in Figure 20-2. The changes you make to the repository are not persistent and will be lost when the application is stopped or restarted.



ID	Client	Location
0	Alice	Board Room
1	Bob	Lecture Hall
2	Joe	Meeting Room 1
3	Jacqui	Meeting Room 7

Figure 20-2. *Using the example application*

Understanding the Role of RESTful Controllers

The example application is an example of a classic web application. All of the logic in the application exists at the server, contained in C# classes, which makes them easy to manage, test, and maintain. But an application designed in this way can have serious deficiencies with regard to speed, efficiency, and openness.

Understanding the Speed Problem

At the moment, the example application is a *synchronous* web application. When the user clicks the Add button, the browser sends the POST request to the server, waits for a response, and then renders the HTML it receives. During this period, the user can't do anything but wait. The waiting period can be imperceptible during development when the browser and the server are on the same machine; however, deployed applications are subject to real-world capacity limits and delays, and the amount of time that a synchronous application requires the user to wait for a response can be substantial.

A synchronous application won't always have a speed problem. For example, if you are writing a line-of-business application for use in a single location where all the clients are connected by a fast and reliable LAN, then you may not have a problem to solve. On the other hand, if you are writing an application for mobile clients in areas with poor infrastructure, then a speed problem can be substantial.

■ **Tip** Some browsers let you simulate different types of network, which can be a useful tool for seeing whether your users are likely to accept working with a synchronous application for a range of scenarios. Google Chrome, for example, offers a feature called *network throttling*, which is available in the Network section of the F12 developer tools. There is a range of predefined networks available, or you can create your own by specifying the upload and download rates and the latency.

Understanding the Efficiency Problem

The efficiency problem arises from the way that a synchronous web application treats the browser as an HTML rendering engine used only to display the HTML documents sent by the server.

When the user first requests the default URL for the example application, for instance, the HTML document that is sent back contains everything that the browser needs to display the content for the application, including the following information:

- The content relies on the Bootstrap CSS file, which should be downloaded if a cached copy isn't available.
- The content contains a form that is configured to send a `POST` request to the `AddReservation` action.
- The content contains a table whose body contains three populated rows.

The example application is simple, and the initial request results in the server sending about 1.3KB of data to the client. However, when the user submits the form, the client is redirected to the `Index` action again, which results in another 1.3KB of data to reflect the addition of a single table row. The browser had already rendered the form and the table, but these are discarded and replaced with an entirely new representation of what is largely the same content.

You may think that 1.3KB of data isn't much and, of course, you would be right. But if you consider the ratio of useful content to duplicate content, you will see that the vast majority of the data sent to the browser is wasted. And the example application is deliberately simple; few real applications require so little HTML, and the amount of duplicated content will be substantially increased as the complexity of the application rises.

Understanding the Openness Problem

The final problem presented by traditional web applications is that the design is closed, meaning that the data in the model can be accessed only through the actions provided by the `Home` controller. Closed applications become a problem when there is a need to use the underlying data in another application, especially when that application is being developed by a different team or even a different organization. Developers often believe that the value in an application is in the user interactions that it offers users, largely because those are the parts that we spend time thinking about and writing. But once an application is established and has an active user base, it is often the data that the application contains that becomes important.

Introducing REST and API Controllers

An *API controller* is an MVC controller that is responsible for providing access to the data in an application without encapsulating it in HTML. This allows the data in the model to be retrieved or modified without having to use the actions provided by the regular controllers, such as the `Home` controller in the example application.

The most common approach for delivering data from an application is to use the *Representational State Transfer* pattern, known as REST. There is no detailed specification for REST, which leads to a lot of different approaches that fall under the RESTful banner. There are, however, some unifying ideas that are useful in client-side web application development.

The core premise of a RESTful web service is to embrace the characteristics of HTTP so that request methods—also known as *verbs*—specify an operation for the server to perform, and the request URL specifies one or more data objects to which the operation will be applied.

As an example, here is a URL that might refer to a specific `Reservation` in the example application:

```
/api/reservations/1
```

The first part of the URL—`api`—is used to separate out the data part of the application from the standard controllers that generate HTML. The next part—`reservations`—indicates the collection of objects that will be operated on. The final part—`1`—specifies an individual object within the `reservations` collection. In the example application, it is the value of the `ReservationId` property that uniquely identifies an object and that would be used in the URL.

URLs that identify an object are combined with HTTP methods to specify operations. In Table 20-3, I have listed the most common HTTP methods and what they represent when combined with an example URL. I have also included details of what data—the *payload*—is included in the request and response for each method and URL combination. The API controller that processes these requests uses the response status code to report on the outcome of the request.

Table 20-3. *Combining HTTP Methods with URLs to Specify a RESTful Web Service*

Verb	URL	Description	Payloads
`GET`	`/api/reservations`	This combination retrieves all the objects.	This response contains the complete collection of `Reservation` objects.
`GET`	`/api/reservations/1`	This combination retrieves the reservation object whose `ReservationId` is 1.	The response contains the specified `Reservation` object.
`POST`	`/api/reservation`	This combination creates a new `Reservation`.	The request contains the values for the other properties required to create a `Reservation` object. The response contains the object that was stored, ensuring that the client receives the saved data.
`PUT`	`/api/reservation`	This combination replaces an existing `Reservation`.	The request contains the values required to change the properties of the specified `Reservation`. The response contains the object that was stored, ensuring that the client receives the saved data.
`PATCH`	`/api/reservation/1`	This combination modifies the existing `Reservation` object whose `ReservationId` is 1.	This request contains a set of modifications that should be applied to the specified `Reservation` object. This response is a confirmation that the changes have been applied.
`DELETE`	`/api/reservation/1`	This combination deletes the `Reservation` object whose `ReservationId` is 1.	There is no payload in the request or response.

Following the RESTful convention isn't a requirement, but it does help make your application easier to work with because the same broad approach has been adopted by many established web applications.

Creating an API Controller

The process for creating an API controller builds on the approach used for standard controllers, with some additional features to help specify the API that is presented to clients. To demonstrate, I added a class file called `ReservationController.cs` to the `Controllers` folder and used it to define the class shown in Listing 20-10. I break down the functionality provided by this controller in the sections that follow.

■ **Tip** Remember that controller classes can be defined anywhere in the project and not just in the `Controllers` folder. For large and complex projects, it can be helpful to define the API controllers separately from the regular HTML controllers and place them in a subfolder or even separate folder entirely.

Listing 20-10. The Contents of the ReservationController.cs File in the Controllers Folder

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;
using Microsoft.AspNetCore.JsonPatch;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ReservationController : Controller {
        private IRepository repository;

        public ReservationController(IRepository repo) => repository = repo;

        [HttpGet]
        public IEnumerable<Reservation> Get() => repository.Reservations;

        [HttpGet("{id}")]
        public Reservation Get(int id) => repository[id];

        [HttpPost]
        public Reservation Post([FromBody] Reservation res) =>
            repository.AddReservation(new Reservation {
                ClientName = res.ClientName,
                Location = res.Location
            });

        [HttpPut]
        public Reservation Put([FromBody] Reservation res) =>
            repository.UpdateReservation(res);

        [HttpPatch("{id}")]
        public StatusCodeResult Patch(int id,
                [FromBody]JsonPatchDocument<Reservation> patch) {
            Reservation res = Get(id);
            if (res != null) {
                patch.ApplyTo(res);
                return Ok();
            }
            return NotFound();
        }

        [HttpDelete("{id}")]
        public void Delete(int id) => repository.DeleteReservation(id);
    }
}
```

API controllers work in the same basic way as regular controllers, which means that you can create a POCO controller or derive a class from the `Controller` base class, which provides more convenient access to the request context data.

ADAPTING THE RESTFUL PATTERN

REST has encouraged a certain amount of dogmatism about how web application APIs should be presented to clients. REST isn't a standard or even a well-defined pattern, and there are some helpful approaches that make REST easier to adopt with an ASP.NET Core MVC application but that have a tendency to upset those programmers who have fixed views about what counts as RESTful.

In Table 20-3, the URLs that I listed for the `POST` and `PUT` operations do not uniquely identify a resource, which some people consider an essential REST characteristic. In the case of the `POST` operation, the unique identifier of a `Reservation` object is assigned by the model, which means that the client is unable to provide it as part of the URL. In the case of the `PUT` operation, the MVC model binding feature—which I describe in Chapter 26 and is the reason I applied the `FromBody` attribute in Listing 20-10—makes it easier to receive details of the `Reservation` object that is to be modified from the request body. So, that's where the `Reservation` controller expects to find the `ReservationId` value that identifies the model object that is to be modified.

In common with all patterns, REST is a starting point that contains helpful and useful ideas. It is not a rigid standard that must be followed at all costs, and the only important thing is to write code that can be understood, tested, and maintained. Accommodating the nature of MVC applications and the design of the repository makes for a simpler application while still providing a useful API for clients to consume. My advice is to consider patterns to be a guiding principle that you adapt to your own needs—something that is as true for REST as it is for MVC itself.

Defining the Route

The route by which API controllers are reached can be defined only using the `Route` attribute and cannot be defined in the application configuration in the `Startup` class. The convention for API controllers is to use a route prefixed with `api`, followed by the name of the controller, so that the `ReservationController` controller shown in Listing 20-10 is reached through the URL `/api/reservation`, like this:

```
...
[Route("api/[controller]")]
public class ReservationController : Controller {
...
```

Declaring Dependencies

API controllers are instantiated in the same way as regular controllers, which means that they can declare dependencies that will be resolved using the service provider. The `ReservationController` class declares a constructor dependency on the `IRepository` interface, which will be resolved to provide access to the data in the model.

```
...
public ReservationController(IRepository repo) => repository = repo;
...
```

Defining the Action Methods

Each action method is decorated with an attribute that specifies the HTTP method that it accepts, like this:

```
...
[HttpGet]
public IEnumerable<Reservation> Get() => repository.Reservations;
...
```

The `HttpGet` attribute is one of a set that is used to restrict access to action methods to requests that have a specific HTTP method or verb. The complete set of attributes is described in Table 20-4.

Table 20-4. *The HTTP Method Attributes*

Name	Description
`HttpGet`	This attribute specifies that the action can be invoked only by HTTP requests that use the `GET` verb.
`HttpPost`	This attribute specifies that the action can be invoked only by HTTP requests that use the `POST` verb.
`HttpDelete`	This attribute specifies that the action can be invoked only by HTTP requests that use the `DELETE` verb.
`HttpPut`	This attribute specifies that the action can be invoked only by HTTP requests that use the `PUT` verb.
`HttpPatch`	This attribute specifies that the action can be invoked only by HTTP requests that use the `PATCH` verb.
`HttpHead`	This attribute specifies that the action can be invoked only by HTTP requests that use the `HEAD` verb.
`AcceptVerbs`	This attribute is used to specify multiple HTTP verbs.

Routes are further refined by including a routing fragment as the argument to the HTTP method attribute, like this:

```
...
[HttpGet("{id}")]
public Reservation Get(int id) => repository[id];
...
```

The routing fragment, `{id}`, is combined with the route defined by the `Route` attribute applied to the controller and a constraint based on the HTTP method. In this case, it means that this action can be reached by sending a `GET` request whose URL matches the `/api/reservations/{id}` routing pattern, where the `id` segment is then used to identify the reservation object that should be retrieved.

Notice that the routes generated for an API controller don't include an `{action}` segment variable, which means that the name of the action method isn't part of the URL required to target a specific method. All the actions in an API controller are reached through the same base URL (`/api/reservation` for the example), and the HTTP method and optional segments are used to differentiate between them.

Defining the Action Results

Action methods for API controllers don't rely on `ViewResult` objects to present their results since there are no views required when delivering data. Instead, API controller action methods return data objects, like this:

```
...
[HttpGet]
public IEnumerable<Reservation> Get() => repository.Reservations;
...
```

This action returns a sequence of `Reservation` objects and leaves MVC to take responsibility for serializing them into a format that can be processed by the client. I explain this process in more detail in the "Understanding Content Formatting" section.

CUSTOMIZING API RESULTS

One of the most appealing aspects of API controllers is that you can just return C# objects from action methods and let MVC figure out what to do with them. MVC is pretty good at working out what to do. For example, if you return `null` from an API controller action method, then the client will be sent a `204 - No Content` response.

But API controllers are able to use the features available to regular controllers, too, and that means you can override the default behavior by returning an `IActionResult` from your action methods that specifies what kind of result you want to send. As an example, here is an implementation of an action method from the example controller that sends a `404 - Not Found` response for queries that don't correspond to an object in the model:

```
...
[HttpGet("{id}")]
public IActionResult Get(int id) {
    Reservation result = repository[id];
    if (result == null) {
        return NotFound();
    } else {
        return Ok(result);
    }
}
...
```

If there is no object in the repository for the specified ID, then I call the `NotFound` method, which creates a `NotFoundResult` object that, in turn, leads to a `404 - Not Found` response being sent to the client. If there is an object in the repository, then I call the `Ok` method to create an `ObjectResult` object. The `Ok` method allows me to send an object to the client within an action that returns an `IActionResult`, as described in Chapter 17. You won't often need to override the default API controller responses, but the full range of action results are available if the need does arise.

Testing an API Controller

There are lots of tools available to help test web application APIs. Good examples include Fiddler (www.telerik.com/fiddler), which is a stand-alone HTTP debugging tool, and Swashbuckle (http://github.com/domaindrivendev/Swashbuckle), which is a NuGet package that adds a summary page to an application that describes its API operations and allows them to be tested.

But the simplest way to make sure that an API controller is to use PowerShell, which makes it easy to create HTTP requests from the command line and which lets you focus on the results of API operations without needing to dig into the details. PowerShell originated on Windows but is now available for Linux and macOS as well.

In the sections that follow, I show you how to use PowerShell to test each of the operations provided by the Reservation controller. You can open a new PowerShell window to run the test commands or use the Visual Studio Package Manager Console window, which uses PowerShell.

Testing the GET Operations

To test the GET operation provided by the Reservation API controller, start the application by selecting Start Without Debugging from the Visual Studio Debug menu and wait until you see the synchronous response provided by the Home controller. Once the application is running, open a PowerShell window and type the following command:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

This command uses the Invoke-RestMethod PowerShell cmdlet to send a GET request to the /api/reservation URL. The result is parsed and formatted to make the data easy to read, as follows:

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Lecture Hall
            2 Joe        Meeting Room 1
```

The server responds to the GET request with a JSON representation of the Reservation objects contained in the model, which the Invoke-RestMethod cmdlet presents in a table format.

UNDERSTANDING JSON

The *JavaScript Object Notation* (JSON) has become the standard data format for web applications. JSON has become popular because it is simple, concise, and easy to work with. It is especially easy to process JSON data in JavaScript code because the JSON format is similar to the way that literal objects are expressed in JavaScript code. Modern browsers include built-in support for generating and parsing JSON data, and popular JavaScript libraries, such as jQuery, will automatically convert to and from JSON.

Although JSON has evolved from JavaScript, its structure is easy for C# developers to read and understand. As an example, here is a response from the API controller in the example application:

```
...
[{"reservationId":0,"clientName":"Alice","location":"Board Room"},
 {"reservationId":1,"clientName":"Bob","location":"Lecture Hall"},
 {"reservationId":2,"clientName":"Joe","location":"Meeting Room 1"}]
...
```

This JSON string describes an array of objects. The array is denoted by the [and] characters, and each object is denoted using the { and } characters. The objects are a collection of key/value pairs, where each key is separated from its value with a colon (the : character) and pairs are separated with commas (the , character). This is loosely similar to the C# literal syntax that I used in the `MemoryRepository` class to define the data in Listing 20-3.

```
...
new List<Reservation> {
    new Reservation { ClientName = "Alice", Location = "Board Room" },
    new Reservation { ClientName = "Bob", Location = "Lecture Hall" },
    new Reservation { ClientName = "Joe", Location = "Meeting Room 1" }
...
```

Notice, however, that MVC changes the capitalization of property names from the C# convention (`ClientName`, with an initial uppercase letter) to the JavaScript convention (`clientName`, with an initial lowercase letter).

Even though the formats are not identical, there is sufficient similarity that a C# developer can read and understand JSON data with little effort. You don't need to get into the detail of JSON for most web applications because MVC does all the heavy lifting, but you can learn more about JSON at `www.json.org`.

There are two GET operations provided by the Reservation controller. When a GET request is sent to /api/reservation, then a response containing all the objects is returned. To retrieve a single object, its ReservationId value is specified as the final segment in the URL, like this:

```
Invoke-RestMethod http://localhost:7000/api/reservation/1 -Method GET
```

This command requests the Reservation object whose ReservationId value is 1 and produces the following result:

```
reservationId clientName location
------------- ---------- --------
            1 Bob        Lecture Hall
```

Testing the POST Operation

All the operations provided by the API controller can be tested using PowerShell, although the format of the commands can be a little awkward. Here is a command that sends a POST request to the API controller to create a new Reservation object in the repository and writes out the data sent back in the response:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method POST -Body
(@{clientName="Anne"; location="Meeting Room 4"} | ConvertTo-Json) -ContentType
"application/json"
```

This command uses the -Body argument to specify the body for the request, which is encoded as JSON. The -ContentType argument is used to set the Content-Type header for the request. The command will produce the following result:

```
reservationId clientName location
------------- ---------- --------
            3 Anne       Meeting Room 4
```

The POST operation uses the clientName and location values to create a Reservation object and returns a JSON representation of the new object to the client, which includes the ReservationId value that has been assigned to the new object. This may seem like the client is simply receiving data values that it has sent to the server in the request, but this approach ensures that the client is working with the same data that the server is using and caters for any formatting or translations that the server performs on the data it receives from the client. To see the effect of the POST request, send another GET request to the /api/reservation API, like this:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

The data that is returned by the client reflects the addition of the new Reservation object.

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Lecture Hall
            2 Joe        Meeting Room 1
            3 Anne       Meeting Room 4
```

Testing the PUT Operation

The PUT method is used to replace existing objects in the model. The ReservationId value of the object is specified as part of the request URL, and the clientName and location values are provided in the request body. Here is a PowerShell command that sends a PUT request to modify a Reservation object:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method PUT -Body
(@{reservationId="1"; clientName="Bob"; location="Media Room"} | ConvertTo-Json)
-ContentType "application/json"
```

This request changes the Reservation object whose ReservationId value is 1 and specifies a new value for the Location property. If you run the command, you will see the following response, which indicates that the change has been made:

```
reservationId clientName location
------------- ---------- --------
            1 Bob        Media Room
```

To see the effect of the PUT request, send a GET request to the /api/reservation API, like this:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

The data that is returned by the client reflects the addition of the new Reservation object.

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            2 Joe        Meeting Room 1
            3 Anne       Meeting Room 4
```

Testing the Patch Operation

The PATCH method is used to modify an existing object in the model. Many applications use PUT requests and ignore PATCH entirely, which is a reasonable approach if your clients have access to all the properties defined by the objects in the model. But in complex applications, clients may receive a specific set of property values for security reasons, which prevents them from sending a complete object as part of a PUT request. PATCH requests are more selective and allow clients to specify a set of granular changes to an object.

ASP.NET Core MVC has support for working with the JSON Patch standard, which allows changes to be specified in a uniform way. I am not going to go into the detail of the JSON Patch standard, which you can read at https://tools.ietf.org/html/rfc6902, but for the example application, the client is going to send the API controller JSON data like this in its HTTP PATCH request:

```
[
 { "op": "replace", "path": "clientName", "value": "Bob"},
 { "op": "replace", "path": "location", "value": "Lecture Hall"}
]
```

A JSON Patch document is expressed as an array of operations. Each operation has an op property, which specifies the type of operation, and a path property, which specifies where the operation will be applied.

For the example application—and, in fact, for most applications—only the replace operation is required, which is used to change the value of a property. This JSON Patch data sets new values for the clientName and location properties, while the object that is to be modified will be identified by the request URL. ASP.NET Core MVC will automatically process the JSON data and present it to the action method as a JsonPatchDocument<T> object, where T is the type of the model object to be modified. The JsonPatchDocument<T> object can then be used to modify an object from the repository using the ApplyTo method. Here is a PowerShell command that sends a PATCH request:

```
Invoke-RestMethod http://localhost:7000/api/reservation/2 -Method PATCH -Body (@
{ op="replace"; path="clientName"; value="Bob"},@{ op="replace"; path="location";
value="Lecture Hall"} | ConvertTo-Json)  -ContentType "application/json"
```

This request asks the server to modify the `clientName` and `location` properties of the `Reservation` object whose `ReservationId` is `2`. To see the effect of the `PUT` request, send a get request to the `/api/reservation` API, like this:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

The data that is returned by the client reflects the addition of the new `Reservation` object.

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            2 Bob        Lecture Hall
            3 Anne       Meeting Room 4
```

Testing the Delete Operation

The final test is to send a `DELETE` request, which will remove a `Reservation` object from the repository, as follows:

```
Invoke-RestMethod http://localhost:7000/api/reservation/2 -Method DELETE
```

The action that accepts `DELETE` requests in the `Reservation` controller doesn't return a result, so no data is displayed when the command has completed. To see the effect of the deletion, request the contents of the repository using the following command:

```
Invoke-RestMethod http://localhost:7000/api/reservation -Method GET
```

The `Reservation` object whose `ReservationId` value is `2` was removed from the repository.

```
reservationId clientName location
------------- ---------- --------
            0 Alice      Board Room
            1 Bob        Media Room
            3 Anne       Meeting Room 4
```

Using the API Controller in the Browser

Defining an API controller has addressed the openness issue for my application, but it hasn't done anything for my speed or efficiency issues. For this, I need to update the HTML part of the application so that it relies on JavaScript to make HTTP requests to the API controller to perform data operations.

In the browser, asynchronous HTTP requests are typically known as *Ajax requests*, where Ajax used to be an acronym for *Asynchronous JavaScript and XML*. The XML data format has lost popularity in recent years, but the name Ajax is still used to refer to asynchronous HTTP requests, even when they return JSON data. More broadly, the technique described in this section is the foundation for single-page applications, where JavaScript in a single HTML page is used to pull in the data for multiple sections of the application, generating the content to display dynamically.

■ **Note** Client-side development is a topic in its own right and outside the scope of this book. In this section, I create only a basic asynchronous HTTP request without detailed explanations, just to give a sense of how it is done. See my *Pro Angular and Essential Angular for ASP.NET Core MVC books, also published by Apress*, for detailed coverage of client-side development with the Angular framework and supporting Angular clients using ASP.NET Core MVC.

There is a JavaScript API provided by browsers for making Ajax requests, but it is a little awkward to deal with, and there are some differences in the way that browsers implement some optional features. The simplest way to make Ajax requests is to use the jQuery library, which is an endlessly useful tool for client-side development. In Listing 20-11, I added the jQuery package to the `bower.json` file.

Listing 20-11. Adding jQuery in the bower.json File in the ApiControllers Folder

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6",
    "jquery": "3.2.1"
  }
}
```

In fact, since some Bootstrap features depend on jQuery, Bower will have already installed the package in the `wwwroot/lib` folder. The addition in Listing 20-11 has the effect of making the dependency explicit. To use the features provided by jQuery, I created the `wwwroot/js` folder and added a JavaScript file called `client.js`, the contents of which are shown in Listing 20-12.

Listing 20-12. The Contents of the client.js File in the wwwroot/js Folder

```
$(document).ready(function () {

    $("form").submit(function (e) {
        e.preventDefault();
        $.ajax({
            url: "api/reservation",
            contentType: "application/json",
            method: "POST",
            data: JSON.stringify({
                clientName: this.elements["ClientName"].value,
                location: this.elements["Location"].value
            }),
            success: function(data) {
                addTableRow(data);
            }
        })
    });
});

var addTableRow = function (reservation) {
    $("table tbody").append("<tr><td>" + reservation.reservationId + "</td><td>"
        + reservation.clientName + "</td><td>"
        + reservation.location + "</td></tr>");
}
```

The JavaScript file in this file responds when the user submits the form in the browser, encodes the form data as JSON, and sends it to the server using an HTTP POST request. The JSON data that is returned by the server is automatically parsed by jQuery and then used to add a row to the HTML table. In Listing 20-13, I have updated the layout to include script elements for the jQuery library for the `client.js` file.

Listing 20-13. Adding JavaScript References in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RESTful Controllers</title>
    <link asp-href-include="lib/bootstrap/dist/css/*.min.css" rel="stylesheet" />
    <script src="lib/jquery/dist/jquery.js"></script>
    <script src="js/client.js"></script>
</head>
<body class="m-1 p-1">
    @RenderBody()
</body>
</html>
```

The first `script` element tells the browser to load the jQuery library, and the second specifies the file that will contain my custom code. There is no obvious visual difference if you run the application and use the HTML form to create a `Reservation` in the application repository, but if you examine the HTTP request that is sent by the browser, you will see that it requires much less data than the synchronous version of the application did. In my simple testing, the asynchronous request required 440 bytes of data, which is about 40 percent of what the synchronous request required. The improvement is more substantial in real applications where the size of data tends to be much less than the size of the HTML document that is used to display it.

Understanding Content Formatting

When an action method returns a C# object as its result, MVC has to work out which data format should be used to encode the object and send it to the client. In this section, I explain what the default process is and how it is influenced by the request sent by the client and the configuration of the application. To help explain how the process works, I added a class file called `ContentController.cs` to the `Controllers` folder and used it to define the API controller shown in Listing 20-14.

Listing 20-14. The Contents of the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object")]
        public Reservation GetObject() => new Reservation {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

I specified static segment variables as the arguments to the `HttpGet` attribute for two of the actions in this controller, which means that they can be reached by the `/api/controller/string` and `/api/controller/object` URLs. The `Content` controller doesn't follow the REST pattern even loosely, but it will make it easy to understand how content negotiation works.

The content format selected by MVC depends on four factors: the formats that the client will accept, the formats that MVC can produce, the content policy specified by the action, and the type returned by the action method. Figuring out how everything fits together can be daunting, but the good news is that the default policy works just fine for most applications, and you only need to understand what happens behind the scenes when you need to make a change or when you are not getting results in the format that you expect.

Understanding the Default Content Policy

The starting point is the standard application configuration that is used when neither the client nor the action method applies any restrictions to the formats that can be used. In this situation, the outcome is simple and predictable.

- If the action method returns a `string`, the string is sent unmodified to the client, and the `Content-Type` header of the response is set to `text/plain`.
- For all other data types, including other simple types such as `int`, the data is formatted as JSON, and the `Content-Type` header of the response is set to `application/json`.

The reason that strings get special treatment is that they cause problems when they are encoded as JSON. When you encode other simple types, such as the C# `int` value `2`, then the result is a quoted string, such as `"2"`. When you encode a string, you end up with two sets of quotes so that `"Hello"` becomes `""Hello""`. Not all clients cope well with this double encoding, so it is more reliable to use the `text/plain` format and sidestep the issue entirely. This is rarely an issue because few applications send `string` values;

it is more common to send objects in the JSON format. You can see both outcomes by using PowerShell. Here is a command that invokes the `GetString` method, which returns a `string`:

```
Invoke-WebRequest http://localhost:7000/api/content/string | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

This command sends a `GET` request to the `/api/content/string` URL and processes the response to display the `Content-Type` header and the content from the response.

■ **Tip** You may receive an error when you use the `Invoke-WebRequest` cmdlet if you have not performed the initial setup for Internet Explorer. This is especially likely on a Windows 10 machine where Edge has replaced it. The problem can be fixed by running IE and selecting the initial configurations you require.

The command produces the following output:

```
Content-Type             Content
------------             -------
text/plain; charset=utf-8 This is a string response
```

The same command can also be used to show the JSON format by changing just the URL that is requested, like this:

```
Invoke-WebRequest http://localhost:7000/api/content/object | select
@{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

This command produces output, formatted for clarity, that shows that the response has been encoded as JSON:

```
Content-Type                   Content
------------                   -------
application/json; charset=utf-8 {"reservationId":100,
                                 "clientName":"Joe",
                                 "location":"Board Room"}
```

Understanding Content Negotiation

Most clients will include an `Accept` header in a request, which specifies the set of formats that they are willing to receive in the response, expressed as a set of MIME types. Here is the `Accept` header that Google Chrome sends in requests:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
```

This header indicates that Chrome can handle the HTML and XHTML formats (XHTML is an XML-compliant dialect of HTML), XML, and the WEBP image format. The `q` values in the header specify relative preference, where the value is 1.0 by default. Specifying a `q` value for 0.9 for `application/xml` tells the server that Chrome will accept XML data but prefers to deal with HTML or XHTML. The final item, `*/*`, tells the server that Chrome will accept any format, but its `q` value specifies that it is the lowest preference of the specified types. Putting all of this together means that the `Accept` header sent by Chrome provides the server with the following information:

1. Chrome prefers to receive HTML or XHTML data or WEBP images.
2. If those formats are not available, then the next most preferred format is XML.
3. If none of the preferred formats is available, then Chrome will accept any format.

You might assume from this that you can change the format a request receives from an MVC application by setting the `Accept` header, but it doesn't work that way—or, rather, it doesn't work that way just yet because there is some preparation required. First, here is a PowerShell command that sends a `GET` request to the `GetObject` method with an `Accept` header that specifies the client will only accept XML data:

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers @{Accept="application/
xml"} | select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

Here are the results, which show that the server has sent an `application/json` response:

```
Content-Type                 Content
------------                 -------
application/json; charset=utf-8 {"reservationId":100,
                              "clientName":"Joe",
                              "location":"Board Room"}
```

Including the `Accept` header has no effect on the format, even though the server has sent the client a format that it hasn't specified. The problem is that, by default, MVC is configured to support JSON only, so it has no other formats it can use. Rather than return an error, MVC sends JSON data in the hope that the client can process it, even though it was not one of the formats specified by the request `Accept` header.

CONFIGURING THE JSON SERIALIZER

ASP.NET Core MVC uses a popular third-party JSON package called Json.Net to serialize objects into JSON. The default configuration is suitable for most projects but can be changed if you need to create JSON in a specific way. The `AddMvc().AddJsonOptions` extension method is used in the `Startup` class and provides access to a `MvcJsonOptions` object, through which the Json.Net package is configured. See `www.newtonsoft.com/json` for details of the configuration options available.

Enabling XML Formatting

To see content negotiation at work, you have to give MVC some choice in the formats it uses to encode response data. Although JSON has become the default format for web applications, MVC can also support encoding data as XML, as shown in Listing 20-15.

■ **Tip** You can create your own content format by deriving from the `Microsoft.AspNetCore.Mvc.Formatters.OutputFormatter` class. This is rarely used because creating a custom data format isn't a useful way of exposing the data in your application and because the most common formats—JSON and XML—are already implemented.

Listing 20-15. Enabling XML Formatting in the Startup.cs File in the ApiControllers Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IRepository, MemoryRepository>();
            services.AddMvc().AddXmlDataContractSerializerFormatters();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

When MVC had only the JSON format available, it had no choice but to encode responses as JSON. Now that there is a choice, you can see the content negotiation process working more fully.

■ **Tip** I used the `AddXmlDataContractSerializerFormatters` extension method in Listing 20-15, but you can also use the `AddXmlSerializerFormatters` extension method, which provides access to an older serialization class. The difference can be helpful if you are generating XML content for older .NET clients.

Here is the PowerShell command that requests XML data again:

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers @{Accept="application/
xml"} | select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

Run this command and you will see that now the server returns XML data, rather than JSON, as follows (I have omitted the XML namespace attributes for brevity):

```
Content-Type                   Content
------------                   -------
application/xml; charset=utf-8 <Reservation>
                                   <ClientName>Joe</ClientName>
                                   <Location>Board Room</Location>
                                   <ReservationId>100</ReservationId>
                               </Reservation>
```

Specifying an Action Data Format

You can override the content negotiation system and specify a data format directly on an action method by applying the Produces attribute, as shown in Listing 20-16.

Listing 20-16. Specifying a Data Format in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object")]
        [Produces("application/json")]
        public Reservation GetObject() => new Reservation {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

The Produces attribute is a filter that changes the content type of ObjectResult objects, which are used behind the scenes by MVC to represent action results in API controllers. The argument for the attribute specifies the format that will be used for the result from the action, and additional allowed types can also be specified. The Produces attribute forces the format used by the response, which can be seen by running the following PowerShell command:

```
(Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/xml"}).Headers."Content-Type"
```

This command displays the value of the Content-Type header from the response to a GET request to the /api/content/object URL. Running the command shows that JSON is used, as specified by the Produces attribute, even though the Accept header of the request specifies that XML should be used.

Getting the Data Format from the Route or Query String

The Accept header isn't always under the control of the programmer who is writing the client, especially if development is being done using an old browser or toolkit. For such situations, it can be helpful to allow the data format for the response to be requested through the route used to target an action method or in the query string section of the request URL. The first step is to define shorthand values in the Startup class that can be used to refer to formats in the route or the query string. There is one mapping by default, in which json is used as shorthand for application/json. In Listing 20-17, I have added an additional mapping for XML.

Listing 20-17. Adding a Format Shorthand in the Startup.cs File in the ApiControllers Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;
using Microsoft.Net.Http.Headers;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IRepository, MemoryRepository>();
            services.AddMvc()
                .AddXmlDataContractSerializerFormatters()
                .AddMvcOptions(opts => {
                    opts.FormatterMappings.SetMediaTypeMappingForFormat("xml",
                        new MediaTypeHeaderValue("application/xml"));
                });
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

The MvcOptions.FormatterMappings property is used to set and manage the mappings. In the listing, I used the SetMediaTypeMappingForFormat method to create a new mapping so that the shorthand xml will refer to the application/xml format. The next step is to apply the FormatFilter attribute to an action method and, optionally, adjust the route for the action so that it includes a format segment variable, as shown in Listing 20-18.

Listing 20-18. Applying the FormatFilter Attribute in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public Reservation GetObject() => new Reservation {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

I have applied the `FormatFilter` attribute to the `GetObject` method and modified the route for the action so that it includes an optional `format` segment. You don't have to use the `Produces` attribute in conjunction with the `FormatFilter` attribute, but if you do, only requests that specify formats for which the `Produces` attribute has been configured will work. Requests that specify a format for which the `Produces` attribute has not been configured will receive a `404 - Not Found` response. If you don't apply the `Produces` attribute, then the request can specify any format that MVC has been configured to use.

I also added the `application/xml` format to the `Produces` attribute so that the action method will support requests for both JSON and XML.

This PowerShell command specifies the `xml` format as part of the request URL:

```
(Invoke-WebRequest http://localhost:7000/api/content/object/xml).Headers."Content-Type"
```

Running this command shows the content type of the response, as follows:

```
application/xml; charset=utf-8
```

The `FormatFilter` attribute looks for a routing segment variable called `format`, gets the shorthand value that it contains, and retrieves the associated data format from the application configuration. This format is then used for the response. If there is no routing data available, then the query string is inspected as well. Here is a PowerShell command that requests XML using the query string:

```
(Invoke-WebRequest http://localhost:7000/api/content/object?format=xml).Headers.
"Content-Type"
```

The format found by the FormatFilter attribute overrides any formats specified by the Accept header, which puts the format selection in the hands of the client developer, even when working with toolkits and browsers that don't allow the Accept header to be set.

Enabling Full Content Negotiation

For most applications, sending JSON data when there is no other format available is a sensible policy since a web application client is more likely to have incorrectly set its Accept header than be unable to process JSON. That said, some applications will have to deal with clients that cause problems if JSON is returned regardless of what the Accept headers say. Getting content negotiation working requires two configuration changes in the Startup class, as shown in Listing 20-19.

Listing 20-19. Enabling Full Content Negotiation in the Startup.cs File in the ApiControllers Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ApiControllers.Models;
using Microsoft.Net.Http.Headers;

namespace ApiControllers {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IRepository, MemoryRepository>();
            services.AddMvc()
                .AddXmlDataContractSerializerFormatters()
                .AddMvcOptions(opts => {
                    opts.FormatterMappings.SetMediaTypeMappingForFormat("xml",
                        new MediaTypeHeaderValue("application/xml"));
                    opts.RespectBrowserAcceptHeader = true;
                    opts.ReturnHttpNotAcceptable = true;
                });
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseStatusCodePages();
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

The RespectBrowserAcceptHeader option is used to control whether the Accept header is fully respected. The ReturnHttpNotAcceptable option is used to control whether a 406 - Not Acceptable response will be sent to the client if there is no suitable format available.

I also have to remove the Produces attribute from the action method so that the content negotiation process isn't overridden, as shown in Listing 20-20.

Listing 20-20. Removing the Produces Attribute in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {

    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        //[Produces("application/json", "application/xml")]
        public Reservation GetObject() => new Reservation {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };
    }
}
```

Here is a PowerShell command that sends a GET request to the /api/content/object URL with an Accept header that specifies a content type that the application cannot provide:

```
Invoke-WebRequest http://localhost:7000/api/content/object -Headers
@{Accept="application/custom"}
```

If you run this command, you will see that the 406 error message is displayed, indicating to the client that the server has been unable to provide the requested format.

Receiving Different Data Formats

When the client sends data to the controller, such as in a POST request, you can specify different action methods to handle specific data formats using the Consumes attribute, as shown in Listing 20-21.

Listing 20-21. Handling Different Data Formats in the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ApiControllers.Models;

namespace ApiControllers.Controllers {
    [Route("api/[controller]")]
    public class ContentController : Controller {

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        //[Produces("application/json", "application/xml")]
        public Reservation GetObject() => new Reservation {
            ReservationId = 100,
            ClientName = "Joe",
            Location = "Board Room"
        };

        [HttpPost]
        [Consumes("application/json")]
        public Reservation ReceiveJson([FromBody] Reservation reservation) {
            reservation.ClientName = "Json";
            return reservation;
        }

        [HttpPost]
        [Consumes("application/xml")]
        public Reservation ReceiveXml([FromBody] Reservation reservation) {
            reservation.ClientName = "Xml";
            return reservation;
        }
    }
}
```

The ReceiveJson and ReceiveXml actions both accept POST requests, and the difference between them is the data format that is specified with the Consumes attribute, which examines the Content-Type header to work out whether the action method can process the request. The result is that when there is a request whose Content-Type is set to application/json, the ReceiveJson method will be used, but if the Content-Type header is set to application/xml, then the ReceiveXml method will be used.

Summary

In this chapter, I explained the role that an API controller plays in an MVC application. I demonstrated how to create and test an API controller, briefly demonstrated how to make asynchronous HTTP requests using jQuery, and explained the content formatting process. In the next chapter, I explain how views and view engines work in more detail.